Freescale Semiconductor
Application Note

Document Number: AN2205
Rev. 2, 01/2007

Car Door Keypad Using LIN

by: Peter Topping
East Kilbride, Scotland

1 Introduction

Car door keypads have traditionally comprised of micro-switches that directly carry the motor currents used to operate windows and mirrors. Although this type of circuit has to handle currents of over an amp, the requirement to drive multiple motors in both directions from more than one switch makes them quite complex (e.g. the control of the front passenger window from both front doors). This included versatility will restrict the possibility of causing a short if two keys are operated at the same time in different directions. In the case of electrically operated folding mirrors, a single multiple contact switch unit controls six motors in both directions. This complexity combined with the high currents resultes in a heavy, expensive, and difficult to install wiring harness, especially in the driver's door. The amount of copper at the hinge where the harness interfaces with the main body wiring can also pose a difficult design compromise between durability and ease of use.

Contents

freescale™ semiconductor

Using a serial multiplex bus like CAN or LIN can resolve these problems. The LIN (Local Interconnect Network, reference 1) bus is lower in cost and ideally suited to use within a door. There are many approaches to designing a LIN based door. CAN or LIN can communicate between the door and the car body. In the latter case, this could use the LIN bus employed within the door or a completely separate bus. If the information comes into the door on a LIN bus, only a single LIN node resides within the door. This minimizes the electronics, but still requires wiring to the mirror, window, lock and keypad modules. Incorporating four separate LIN nodes with only three wires interconnecting them reduces the wiring to a minimum. There is only one LIN data line, the other two connections are the positive and negative supplies. This application note considers the design of the keypad module required to facilitate this type of design. It has window, mirror, and child-lock functionality and communicates directly (or via a controller in the car) with the window and mirror modules. Lock control is not included as it would normally come directly from the body controller, but a main lock key could be incorporated if required.

The described implementation of the keypad is a LIN slave node so it cannot initiate communications with other nodes. This is the responsibility of the LIN master, in this case the body controller. On a regular basis, say every 100ms, the master issues the appropriate message to the keypad. It responds with a four-byte response formatted as shown in table 1. Depending on the firmware in the other nodes, they can directly read this information (slave to slave communication) or read by the master and retransmit to the individual nodes.

Table 1. Format of Keypad Output Data

ID $20	Front Windows (byte 0)	Rear Windows (byte 1)	Mirrors (byte 2)	Miscellaneous. (byte 3)
bit 0	Driver, Express UP	D. side, Express UP	Driver, UP	LIN – bit error
bit 1	Driver, Express DOWN	D. side, Express DOWN	Driver, DOWN	LIN – checksum error
bit 2	Driver, Manual UP	D. side, Manual UP	Driver, LEFT	LIN – identifier parity error
bit 3	Driver, Manual DOWN	D. side, Manual DOWN	Driver, RIGHT	LIN – slave not responding error
bit 4	Pass., Express UP	P. side, Express UP	Passenger, UP	LIN – inconsistent sync. error
bit 5	Pass., Express DOWN	P. side, Express DOWN	Passenger, DOWN	LIN – no bus activity error
bit 6	Pass., Manual UP	P. side, Manual UP	Passenger, LEFT	Mirror fold
bit 7	Pass., Manual DOWN	P. side, Manual DOWN	Passenger, RIGHT	Disable rear funct. (child lock)

2 Mirror keys

A car door differs from a calculator or a phone in that it makes sense to press more than one key at a time. This renders the traditional approach to keypad design, an X-Y matrix, of limited use in automotive applications. Without additional components, a matrix has only a limited capability to decode multiple key presses. If two keys are pressed on the same row or the same column, two lines are shorted together and the determination of the unique identity of a third pressed key lessens. The recognition of only one up, down, left, and right allows an acceptable employment of a matrix of mirror keys. The design presented here uses this arrangement for the eight mirror functions required by two mirrors. The folding key adds a ninth (ten if a separate un-fold key were required) and the resultant matrix becomes 5x2. This facilitates

nine (or 10) functions using seven I/O lines, a small but perhaps significant saving over using separate lines for each function.

Figure 1 shows the simplest, and reasonably I/O efficient, arrangement. This employs six lines whereby four provide up, down, left, and right, with the two additional lines used for fold and driver/passenger mirror selection. In this case, a simple slide switch with a dedicated I/O line for the mirror selection makes a 3x2 matrix unnecessary. There is no advantage using such a small matrix (2+2=2x2) for the four movement keys, and it would have the disadvantage mentioned above. Other approaches could reduce the I/O requirement further. Coding on four lines would give 16 possibilities, more than the 10 actually required (4 directions on 2 mirrors, fold and nothing). Simpler coding could employ five lines to provide mirror select, move, fold, and two binary lines for direction. In the extreme case, the use of an analog-to-digital input and a few resistors would allow control of the mirrors by a single line into the keypad MCU.

The actual solution presented here used an existing joystick which dictated the use of a 5x2 matrix (and hence seven I/O lines). The two columns could constitute driver/passenger mirror selection, but the particular joystick used provided the different arrangement shown in figure 2. Functionality like mirror position store was not incorporated in this application and may, depending on implementation, require additional keys.

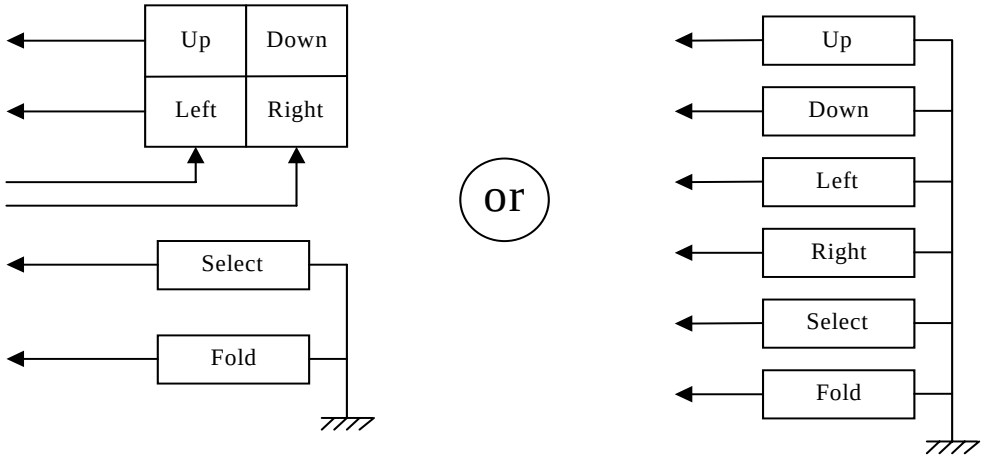

Figure 1. Simple Mirror Key Implementation.

3 Window keys

The keypad requirements for window control are different from those of the mirror in that the driver door may have keys for all four windows, and it is possible, and not unreasonable, to expect two or more to operate simultaneously. For this reason, although there is a possible requirement of eight or even six keys, a conventional key matrix is not necessary. Even though up and down for four windows requires eight keys, further travel on a key causes an “express” movement of the window whereby it travels fully up (or down) even with the key’s release. The simplest way to implement this is to have 16 switches. As a 4x4 matrix is not appropriate without many additional components this would use 16 I/O lines from the controlling MCU. This number may require a more expensive device due to its availability or forcing the use of a higher pin-count. Table 2 shows some approaches to this problem.

Table 2. . Possible Window Key Arrangements

Method	Advantages	Disadvantages
Direct interface to 16 switches	Simple software	Uses 16 I/O lines
Use analog-to-digital inputs	Uses only 4 I/O lines	Requires an MCU with analog-to-digital and some external components
8 switches where the duration of press distinguishes between a normal and an express action.	Uses only 8 I/O lines	Poorly defined and confusing user interface
8 switches where an “Express UP” is requested by also closing the DOWN contact etc.	Uses only 8 I/O lines	More complex sequence-dependent software

The second method would employ four contacts per window and use resistors to provide five different output voltages corresponding to “Express up”, “Up”, no action, “Down” and “Express down”. As even an 8-bit analog-to-digital with a ±2 bit accuracy could distinguish between 64 levels, this requires little voltage accuracy and possible use of low precision resistors. As with the first method, implementation in software is not difficult and these two methods are not considered further in this application note.

The next often used method has no standard as to whether it is a long or a short press that results in an express movement. While it would seem more logical to have the long press cause an express movement, moving the window to an intermediate position requires at least two presses (one long press to start an express movement and a second short one to stop it) making it the poorer choice. If the less intuitive choice is made and a short press initiates an express movement, then the window moves to an intermediate position by holding the key down until the desired position is reached. The time method is now less popular presumably because of the confusion caused by the lack of consistency among manufacturers when making this choice.

The fourth method is the solution presented here. Figure 2 illustrates one design for the type of switch that has separate contacts for up and down, but no additional contacts for express up and express down. When either of the extreme positions (express up or express down) is selected, the floating bar in the switch causes the second micro-switch to activates. In both cases, micro-switches are pressed and the only method of distinguishing between express up and express down is to determine which switch was activated first.

		Contact	
		Up	Down
Switch position	Express up	*	*
	Up	*	
	Off		
	Down		*
	Express down	*	*

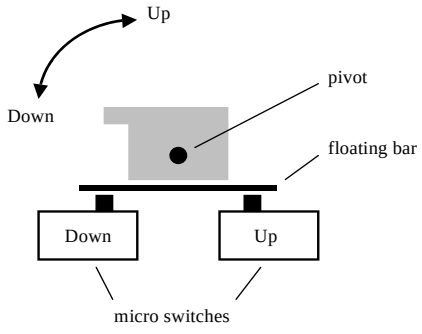

Figure 2. 2-Contact Window Switch.

Although not a complex application, you must take care to independently interpret the sequence of key presses of the four windows while incorporating a reasonable immunity to spurious noise and contact bounce. As a double closure caused by contact bounce would not necessarily cause a problem in this application, a possibility still exists to omit any kind of contact debounce filtering. However, this leaves the keyboard open to responding incorrectly in the presence of noise and 15ms of debounce was incorporated. The double contact nature of the operation dictates quite short debounce time because if the second contact were closed before the first one had been fully debounced, then the first press would not be recognized. This would end up with the controlling MCU knowing about a required express action, but not knowing which direction to go as it is unaware of the first pressed key.

4 Hardware

The target MCU for the keypad module is the MC68HC908EY16. As this MCU is not available at the time of writing this application note it currently uses an MC68HC908AZ60A. Implementation on an MC68HC908EY16 or EY8 would significantly reduce the cost. Not only is the MC68HC908EY16 a lower pin-count lower cost device, but it will include an on-chip ICG (internal clock generator) module obviating the need for a crystal or ceramic resonator.

Figure 3 shows the circuit diagram used in the keyboard application. Apart from the MCU itself, two chips are required to implement a simple LIN node. These are the LIN interface, in this case the Freescale MC33399 (or MC33661) and a 5 volt regulator. A single chip, the MC33689 (LIN SBC), replaces these two chips. The regulator used was the 8-pin LT1121 which has the capability of shutting down into a low power sleep mode under the control of the MCU. In the arrangement, the MCU or the MC33399 (or MC33661) can wake it.

The MC33399 (or MC33661) includes the 30kohm LIN pull-up so this does not need to be included on the PCB. The only discrete components required are pull-up resistors for IRQ, Reset and the port pins used for the window keys, decoupling capacitors and a crystal, and its associated components.

This type of application typically includes a child-lock key which inhibits the operation of the rear windows using their local keys. A pull-up for the I/O line used for this purpose and an LED to indicate that this function is activated are also included. A driver for keypad illumination was also incorporated onto the PCB so that the LIN bus could control it. PortC pull-ups and an IRQ jumper to 9 volts were also added to facilitate entry into monitor mode using an external serial interface. This facilitated in-circuit programming of the on-chip flash memory. The software was developed on the prototype PCB fitted with a target header for the MMDS development system rather than with an actual MC68HC908AZ60A.

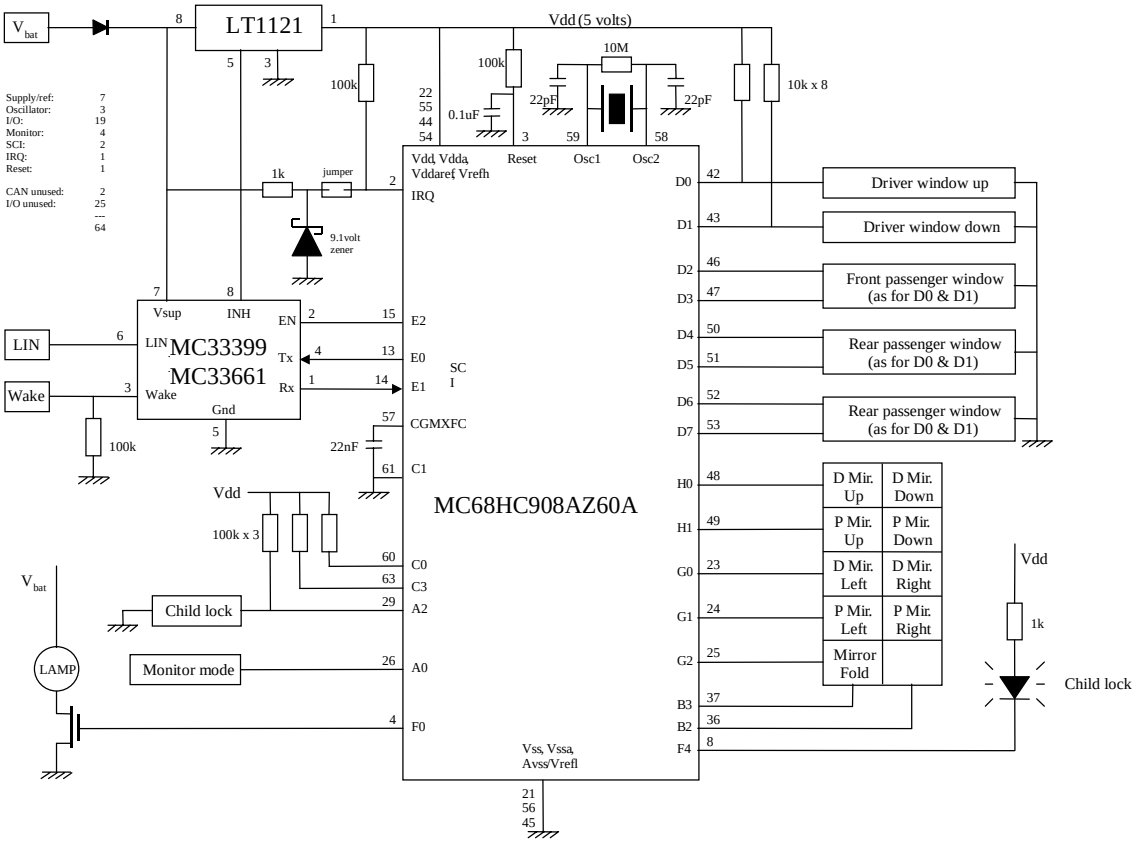

Figure 3. Keypad Module Circuit Diagram

5 Software

The keypad module uses the Freescale/Metrowerks LIN driver software so the I/O activity is handled without the application code which simply uses a “LIN_PutMsg()” call to update the buffer used by the drivers. Similarly, a “LIN_GetMsg()” is used for receiving data. The main purpose of the keypad module is to supply data for the window and mirror modules and the only feature of the module controlled by “LIN_GetMsg()” is the keypad illumination. The use of the LIN drivers leaves the programmer free to think about the application without having to worry about the communications protocol.

The main loop uses the programmable interrupt counter to poll the keys at 200Hz. The eight lines from the window keys are read from portD. The mirror keys must read the two columns of the matrix separately using portG and portH. As there is no need to recognise a multiple keypress, this is done by checking the first column and, only if no key is pressed, addressing (using portB, bit3) and checking the second column. Although the MC68HC908AZ60A’s keyboard interrupt feature is not used, the I/O lines with this capability employ it to allow a keypress initiated wake-up without changing the hardware. This dictates that portH and portG are read. If a key is pressed, its column adds to the result to form a mirror byte that returns the overall state of the mirror keys. Once read, the state of the child lock switch adds to the “mirror” byte resulting in a 2-byte keypad status as shown in table 3.

Table 3. Format of Keypad Status Bytes

Window byte	D7	D6	D5	D4	D3	D2	D1	D0
Mirror byte	Child lock	-	Column	G2 (fold)	G1	G0	H1	H0

Comparing these two bytes with their status the previous time around the loop carries the keyboard debounce. If they appear the same on three consecutive reads, then it uses the status to determine the data sent by the keypad module. Figure 3 shows the flow diagram for this part of the code. In the flow diagram, the keypad “status” refers to the result of a single read of all the keys and the keypad “set-up” to a confirmed reading of the keys, i.e. the same result of the status on three consecutive reads.

Select the key debounce time by changing the number the variable count is compared with. In this application the value of one corresponds to the requirement for three consecutive identical reads before any change is recognised. If two consecutive reads are different, then count is cleared and the new status is saved. A second read 5ms later which gives the same result increments count to one. A third identical read sees that count is one and transfers the confirmed status into a valid key set-up. Before this is done, the previous set-up saves for a possible correct interpretation of the window keys. At this point, count increments again and its value of two used to prevent any further increments as the loop contents repeatedly execute. Without this locking of the value of count, it would wrap around causing unnecessary updates of the LIN buffer.

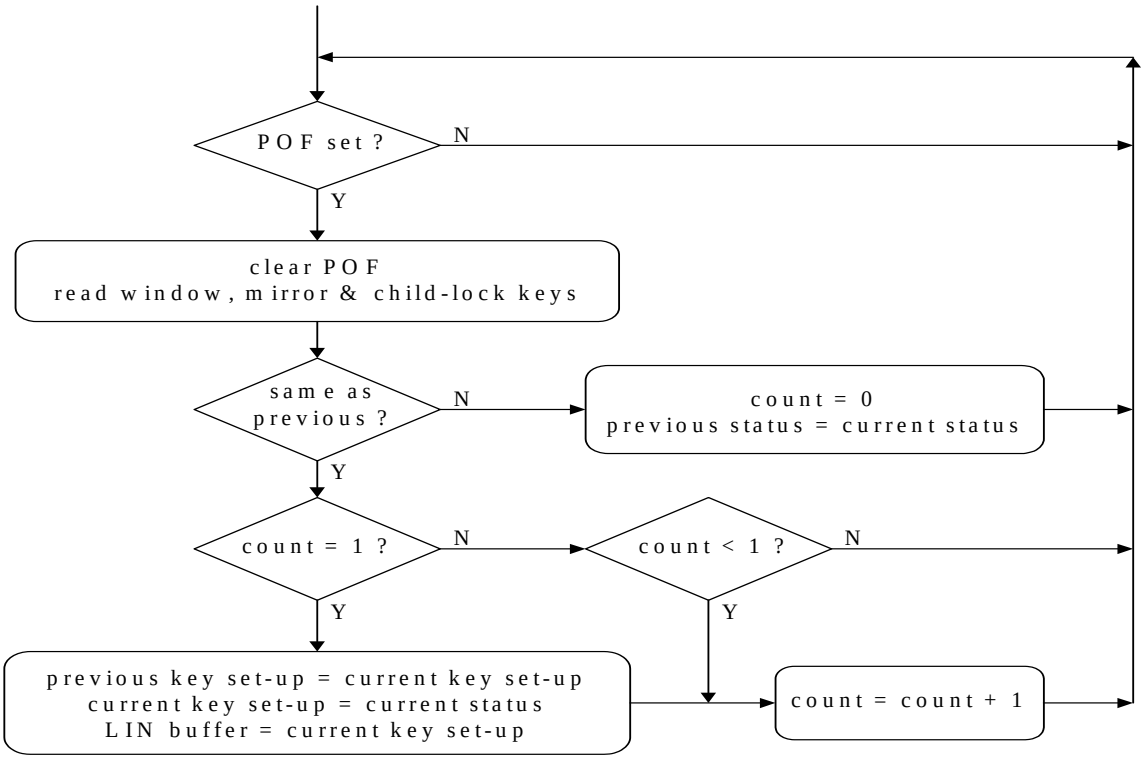

Figure 4. Flow Chart of Main Software Loop

The other four functions in the code (see appendix for a listing of the code) are called if the decision updates the LIN buffer. “Save_history()” executes first out of these functions. This function updates the variable window_old with the previous set-up to allow a comparison when a new key set-up becomes valid. The requirement that the previous state of the keys for a window remain because something has changed on another window complicates this operation. For this reason, update the two bits corresponding to the two keys for each window only if a change occurs in either of these bits after the previous time around the loop. This is performed separately for each window to retain their four, completely independent “histories”.

Once the history is saved, the current key set-up updates with the newly confirmed status and the function “Prepare_new_data()” is called. For each window in turn, this function extracts the current and previous up and down bits and, using function “Get_window_bits()”, converts the data into the format required for the LIN message buffer. “Get_window_bits()” uses case statements to convert the bits as shown in table 4.

Table 4. Window Bit Conversion

Up	Down	Previous Up	Previous Down	Move Up	Express Up	Move Down	Express Down
0	0	X	X	0	0	0	0
0	1	0	0	0	0	1	0
0	1	0	1	0	0	1	0
0	1	1	0	0	0	1	0
0	1	1	1	0	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	1	0	0
1	1	1	1	0	0	0	0

The simplest case is when both of the Up and Down bits are zero: no bits are set in the LIN buffer regardless of the values of the previous bits. When only the Down bit is set, the corresponding bit in the LIN buffer is set except when the previous bits are both one. This happens with an Express Down request when the key releases half way to the Down position. As sending a Down command stops the express movement in the window module, this condition continues to set the Express Down bit (clearing all the bits in this circumstance would be a satisfactory alternative as this would also allow the express movement to continue). The treatment of the bits when Up is set and Down clear is similar.

Figure 2 explains the request of an express movement when both the Up and Down bits are set. The appropriate bit is set in the LIN message according to which of the previous up and down bits is set. If,

however, neither or both of the previous bits are set, then it requests no movement as no sufficient information exists to indicate whether an express Up or an express Down is required. These are error conditions which shouldn't occur in practice.

Once the window bits have been interpreted and the decisions placed in bytes 0 & 1 of the LIN buffer (see table 1), the function "Prepare_new_data()" does the same for the mirror using the function "Get_mirror_bits()". After checking that the mirrors are not folded, this function comprises a single case statement which converts the column bit and the four mirror movement bits into the format required by the LIN buffer (byte 2, table 1). No bits are set if there are no input bits set or if there is an illegal combination of input bits (i.e. more than one movement bit set).

Finally, "Prepare_new_data()" checks the mirror fold and child lock bits and, if required, sets them in byte 3 of the buffer. The status of the child-lock bit does not affect the functionality of the keypad module, assuming that the body controller or the window lift modules will read this bit and, if appropriate, inhibit window operation.

The last task of the main loop is to use the LIN driver function "LIN_GetMsg()" to read the 4-byte message available from the body controller LIN master (see table 5). Bit 6 of byte 2 is the only relevant bit used to control keypad illumination. The state of this bit is read and the keypad lamps controlled using a buffer connected to portF bit 0.

Table 5. Format of Body Controller Output Data

ID $21	Locking (byte 0)	Mirrors (byte 1)	Miscellaneous (byte 2)	Miscellaneous (byte 3)
bit 0	Driver, superlock	Store Mirror, MEM 0	Mirror, defrost	LIN – bit error
bit 1	Driver, lock	Store Mirror, MEM 1	Mirror, functionality	LIN – checksum error
bit 2	Passenger, superlock	Store Mirror, MEM 2	Window, front enabled	LIN – identifier parity error
bit 3	Passenger, lock	Store Mirror, MEM 3	Window, rear enabled	LIN – slave not resp. error
bit 4	Rear D. side, superlock	Recall Mirror, MEM 0	Window, central opening	LIN – inconsistent sync. error
bit 5	Rear D. side, lock	Recall Mirror, MEM 1	Window, central closing	LIN – no bus activity error
bit 6	Rear P. side, superlock	Recall Mirror, MEM 2	Front switches, lights	Door flasher, lights
bit 7	Rear P. side, lock	Recall Mirror, MEM 3	Rear switches, lights	Door position, lights

6 References

1. LIN Protocol Specification, Version 1.2, 17 November 2000.
2. MC68HC908AZ60A Technical Data.
3. MC68HC908EY16 Advance Information.

Appendix A Software Listing

```
/*****************************************************************************
*
*               "DNA" door project - Keyboard module
*               ============================================
*
*    Originator:    P. Topping
*    Date:          22nd June 2001
*    Modified:      13th July 2001
*    Comment:       Changed PIT divide ratio for 200Hz @ 8MHz (for 9600baud).
*                   Changed UP & DOWN to EXPRESS if previous was EXPRESS.
*                   Reduced debounce to 3 (10-15ms).
*                   Added keypad illumination control.
*
*****************************************************************************/

/*****************************************************************************
*
*    Header file includes
*
*****************************************************************************/

#include <hc08az60.h>
#include <linapi.h>


/*****************************************************************************
*
*    Function prototypes
*
*****************************************************************************/

unsigned char Get_window_bits (unsigned char, unsigned char);
unsigned char Get_mirror_bits (void);
void Prepare_new_data (void);
void Save_history (void);


/*****************************************************************************
*
*    Globals
*
*****************************************************************************/

unsigned char window;
unsigned char mirror;
unsigned char window_cur;
unsigned char mirror_cur;
unsigned char window_old;

unsigned char Kpm_data[4];
unsigned char Master_data[4];
```

```
/*****************************************************************************
* Function:        LIN_Command
*
* Description:     User call-back.
*                  Called by the driver after successful transmission or receiving
*                 of the Master Request Command Frame (ID Field value '0x3C').
*
* Returns:         never returns
*
*****************************************************************************/

void LIN_Command()
{
    while(1)
    {
    }
}

/*****************************************************************************
*
*    Function name: Main
*    Originator:    P. Topping
*    Date:          5th June 2001
*
*****************************************************************************/

void main (void)
{

unsigned char count = 0;

unsigned char window_last;
unsigned char mirror_last;

CONFIG1 = 0x71;
CONFIG2 = 0x19;

DDRA    = 0xF0;
DDRB    = 0xFF;
DDRC    = 0x34;
DDRD    = 0x00;
DDRE    = 0xDD;
DDRF    = 0x7F;
DDRG    = 0x00;
DDRH    = 0x00;

PTA     = 0x00;
PTB     = 0x00;
PTC     = 0x00;
PTE     = 0x04;                                  /* MC33399 enable high     */
PTF     = 0x00;

Kpm_data[0] = 0;
Kpm_data[1] = 0;
Kpm_data[2] = 0;
Kpm_data[3] = 0;

asm CLI;

LIN_Init();
```

```
LIN_PutMsg (0x20, Kpm_data);

PITSC = 0x10;                                   /* start PIT at /1         */

PMODH = 0x27;                                   /* /10000 for a repetition */
PMODL = 0x10;                                   /* rate of 200Hz @ 8MHz.   */


while (1)
{
    if (PITSC & 0x80)                           /* is PIT overflow set?    */
    {
        PITSC &= ~(0x80);                       /* yes, clear it           */

        window = ~PTD;                          /* read window port        */

        PTB = 0x04;                             /* mirror, column 1        */
        mirror = ((~PTH & 0x03) | (0x04*(~PTG & 0x07)));      /* read rows   */
        if (mirror == 0)                        /* key pressed?            */
        {
            PTB = 0x08;                         /* no, try column 2        */
            mirror = ((~PTH & 0x03) | (0x04*(~PTG & 0x07)));  /* read rows   */
            if (mirror)                         /* key pressed?            */
            {
                mirror = mirror |= 0x20;        /* yes, set 2nd column bit */
            }
        }

        if ((0x04 & PTA) == 0x04)               /* child lock active?      */
        {
            mirror = mirror |= 0x80;            /* yes, add MSbit          */
        }

        if ((mirror == mirror_last) && (window == window_last))   /* same ?  */
        {
            if (count == 1)                     /* yes, third time ?       */
            {
                Save_history();                 /* yes, save prev. window  */
                mirror_cur = mirror;            /* set-up and xfer new     */
                window_cur = window             /* status into current     */;
                Prepare_new_data ();            /* xfer data to LIN buffer */
                count ++;                       /* count=2 stops re-entry  */
            }
            else if (count < 1)                 /* count=2 stops increment */
            {
                count ++;
            }
        }
        else
        {
            count = 0;                          /* no, different, so reset */
            mirror_last = mirror;               /* count and save current  */
            window_last = window;               /* status as last status   */
        }

        LIN_GetMsg (0x21, Master_data);
        if ((Master_data[2] & 0x40) == 0x40)
        {
            PTF |= 0x01;                        /* lights on               */
        }
```

```
        else
        {
            PTF &= ~(0x01);                         /* lights off              */
        }
    }
}
}
```

```
/*****************************************************************************
*
*    Function name: Prepare_new_data
*    Originator:    P. Topping
*    Date:          14th June 2001
*    Modified:
*    Purpose:       Put new mirror and window status into buffer.
*    Parameters:    input:  none
*                   output: none
*    Comment:
*
*****************************************************************************/

void Prepare_new_data (void)
{

    unsigned char bits;
    unsigned char up_down;
    unsigned char up_down_old;

    up_down = (0x03 & (window_cur >> 6));          /* Driver's window         */
    up_down_old = (0x03 & (window_old >> 6));
    Kpm_data[0] = Get_window_bits(up_down, up_down_old);

    up_down = (0x03 & (window_cur >> 4));          /* Front pass. window      */
    up_down_old = (0x03 & (window_old >> 4));
    bits = Get_window_bits(up_down, up_down_old);
    Kpm_data[0] |= bits << 4;                      /* top nibble              */

    up_down = (0x03 & (window_cur >> 2));          /* Rear window (driver)    */
    up_down_old = (0x03 & (window_old >> 2));
    Kpm_data[1] = Get_window_bits(up_down, up_down_old);

    up_down = (0x03 & window_cur);                 /* Rear window (pass.)     */
    up_down_old = (0x03 & window_old);
    bits = Get_window_bits(up_down, up_down_old);
    Kpm_data[1] |= bits << 4;                      /* top nibble              */

    Kpm_data[2] = Get_mirror_bits();               /* mirror byte             */

    bits = 0;                                      /* bits 0-6 not used       */
    if (mirror_cur & 0x10)
    {
        bits = 0x40;                               /* fold mirrors            */
        PTF &= ~(0x02);                            /* fold LED on (debug)     */
    }
    else
```

```
    {
        PTF |= 0x02;                            /* fold LED off (debug)    */
    }
    if (mirror_cur & 0x80)
    {
        bits |= 0x80;                           /* disable rear functions  */
        PTF &= ~(0x10);                         /* child-lock LED on       */
    }
        else
    {
        PTF |= 0x10;                            /* child-lock LED off      */
    }
    Kpm_data[3] = bits;                         /* misc. byte              */

    LIN_PutMsg (0x20, Kpm_data);                /* update LIN buffer       */
}
```

```
/*****************************************************************************
*
*    Function name: Save_history
*    Originator:    P. Topping
*    Date:          12th June 2001
*    Modified:
*    Purpose:       Check each window for a change. If either bit has changed
*                   then both bits are saved, otherwise neither is saved.
*    Parameters:    input:  none
*                   output: none
*    Comment:
*
*****************************************************************************/

void Save_history (void)
{
    if (((window_cur & 0x01) != (window & 0x01)) |
        ((window_cur & 0x02) != (window & 0x02)))
    {
        window_old = (window_old & ~0x03) | (window_cur & 0x03);
    }

    if (((window_cur & 0x04) != (window & 0x04)) |
        ((window_cur & 0x08) != (window & 0x08)))
    {
        window_old = (window_old & ~0x0C) | (window_cur & 0x0C);
    }

    if (((window_cur & 0x10) != (window & 0x10)) |
        ((window_cur & 0x20) != (window & 0x20)))
    {
        window_old = (window_old & ~0x30) | (window_cur & 0x30);
    }

    if (((window_cur & 0x40) != (window & 0x40)) |
        ((window_cur & 0x80) != (window & 0x80)))
    {
        window_old = (window_old & ~0xC0) | (window_cur & 0xC0);
    }
}


/*****************************************************
  hc08az60.h
  Register definitions for the 908AZ60
*****************************************************


#define PTA *((volatile unsigned char *)0x0000)
#define PTB *((volatile unsigned char *)0x0001)
#define PTC *((volatile unsigned char *)0x0002)
#define PTD *((volatile unsigned char *)0x0003)
#define PTE *((volatile unsigned char *)0x0008)
#define PTF *((volatile unsigned char *)0x0009)
#define PTG *((volatile unsigned char *)0x000A)
#define PTH *((volatile unsigned char *)0x000B)

#define DDRA *((volatile unsigned char *)0x0004)
#define DDRB *((volatile unsigned char *)0x0005)
#define DDRC *((volatile unsigned char *)0x0006)
```

```
#define DDRD *((volatile unsigned char *)0x0007)
#define DDRE *((volatile unsigned char *)0x000C)
#define DDRF *((volatile unsigned char *)0x000D)
#define DDRG *((volatile unsigned char *)0x000E)
#define DDRH *((volatile unsigned char *)0x000F)

#define CONFIG1 *((volatile unsigned char *)0x001F)
#define CONFIG2 *((volatile unsigned char *)0xFE09)

#define PITSC *((volatile unsigned char *)0x004B)
#define PCNTH *((volatile unsigned char *)0x004C)
#define PCNTL *((volatile unsigned char *)0x004D)
#define PMODH *((volatile unsigned char *)0x004E)
#define PMODL *((volatile unsigned char *)0x004F)

#define VECTF (void(*const)())                        */




/*****************************************************************************
*
*    Function name: Get_window_bits
*    Originator:    P. Topping
*    Date:          7th June 2001
*    Modified:
*    Purpose:       Converts current and previous window key data into format
*                   required for LIN message.
*    Parameters:    input:  current and previous UP and DOWN bits
*                   output: UP, DOWN, EXPRESS UP & EXPRESS DOWN bits.
*    Comment:
*
*****************************************************************************/

unsigned char Get_window_bits (unsigned char up_down, unsigned char up_down_old)
{

    unsigned char bits;

    switch (up_down)
    {
        case 0x00:                                  /* neither                */
            bits = 0;
            break;

        case 0x01:                                  /* down, check previous    */
            switch (up_down_old)
            {
                case 0x03:                          /* was previous express ?  */
                    bits = 0x02;                    /* yes, keep it that way   */
                    break;
                default:                            /* no, just down           */
                    bits = 0x08;
            }
            break;

        case 0x02:                                  /* up, check previous      */
            switch (up_down_old)
            {
```

```
                case 0x03:                          /* was previous express ?  */
                    bits = 0x01;                    /* yes, keep it that way   */
                    break;
                default:                            /* no, just up             */
                    bits = 0x04;
            }
            break;

        default:                                    /* both, check previous    */
            switch (up_down_old)
            {
                case 0x00:                          /* neither (illegal)       */
                    bits = 0;
                    break;
                case 0x01:                          /* down (express)          */
                    bits = 0x02;
                    break;
                case 0x02:                          /* up (express)            */
                    bits = 0x01;
                    break;
                default:                            /* both (illegal)          */
                    bits = 0;
            }
    }
    return bits;
}
```

```
/*****************************************************************************
*
*    Function name: Get_mirror_bits
*    Originator:    P. Topping
*    Date:          7th June 2001
*    Modified:
*    Purpose:       Converts current mirror key data into format required for
*                   LIN message.
*    Parameters:    input:  mirror_cur (global)
*                   output: Driver/passenger Up, Down, Left & Right
*    Comment:
*
*****************************************************************************/

unsigned char Get_mirror_bits (void)
{

    unsigned char bits;

    if (mirror_cur & 0x10)
    {
        return 0;                       /* mirrors folded, clear move bits    */
    }
    else
    {
        switch (mirror_cur & 0x2F)
        {
            case 0x01:                  /* driver mirror up                   */
                bits = 0x01;
                break;
            case 0x21:                  /* driver mirror down                 */
                bits = 0x02;
                break;
            case 0x02:                  /* passenger mirror up                */
                bits = 0x10;
                break;
            case 0x22:                  /* passenger mirror down              */
                bits = 0x20;
                break;
            case 0x04:                  /* driver mirror left                 */
                bits = 0x04;
                break;
            case 0x24:                  /* driver mirror right                */
                bits = 0x08;
                break;
            case 0x08:                  /* passenger mirror left              */
                bits = 0x40;
                break;
            case 0x28:                  /* passenger mirror right             */
                bits = 0x80;
                break;
            default:                    /* none (or > 1 bits) set, clear all  */
                bits = 0;
        }
    }
    return bits;
}
```

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN2205
Rev. 2
01/2007

freescale™
semiconductor